Python Grundlagen

Julian Huber & Matthias Panny

Grundlegende Syntax

🎯 Lernziele

- Studierenden können Variablen zuweisen, auswerten und Kontrollstrukturen definieren
- Studierende können einfache Algorithmen mit Python umsetzen

Sequenz und Kommentare

- Code wird zeilenweise interpretiert
- Wenn Sie Code direkt im Interpreter eingeben, wird das Ergebnis der letzten Zeile auch ohne `print()` ausgegeben → REPL
- Kommentare werden nicht interpretiert

Kommentare

- Einzeilige Kommentare mit `#`

```
# This is a single-line comment
1+2
# > 3
```

- Mehrzeilige Kommentare mit `"""` oder `'''`

```
"""
This is a
multi-line comment
"""
1+2
# > 3
```

Names

Names

- was über einen Namen (Zeichenkette) aufgerufen werden kann
- Variablen, Funktionen, Klassen, Module, etc.
- wenn unbekannt folgt ein `NameError`

Befehle und Funktionsnamen

- z.B. Teil des Standardumfangs der Sprache oder eigene Funktionen: `print(1)`
- ⚠: solche Schlüsselwörter (`print`, `list`) im Namespace können überschrieben werden → unerwartete Fehler

Variablennamen

- Names können auf Variablen eines bestimmten Datentyps enthalten

```
message = "Hello, Python World!"
print(message)
```

Datentypen

- In C nur statische Typisierung (static typing) kennen gelernt → Datentypen müssen vorher deklariert werden
- Python nutzt dynamische Typisierung (dynamic typing)

Dynamic Typing

- Variablentypen müssen nicht vorher deklariert werden

```
a_variable = 42                 # = als Operator für Zuweisung
print(type(a_variable))         # type() als Befehl für die Ausgabe des Typs
# > <class 'int'>               # Rückgabe
```

- Variablen und können sich dynamisch während der Ausführung verändern

```
a_variable = 'This is a string now.'
type(a_variable)
# > <class 'str'>               # Der Name a_variable enthält nun einen Sting
```

- Es können so genannte Type Hints gesetzt werden → signalisiert den Typ der Variable nur für den Programmierer

```
a_variable: int = 42
```

Datentypen

Datentypen in Python

- Text Type: `str`
- Numeric Types: `int`, `float`, `complex`
- Sequence Types: `list`, `tuple`, `range`
- Mapping Type: `dict` (später mehr)
- Boolean Type: `bool`
- None Type: `NoneType` → z.B. nützlich, wenn Variable angelegt aber noch noch nicht initialisiert wird

🤓 Vollständige Liste an Datentypen

- Python Docs Built-in Types
- Python Docs Datatypes

Datentypen

"NoneType" in C++

- in C++ gibt es keinen expliziten NoneType (aber nullptr für Zeiger)
- Deklaration ohne Initialisierung in C++

```
int myNum; // Declare a variable without initializing it
int* myPtr = nullptr;
```

NoneType in Python

- in Python gibt es den expliziten NoneType
- Initialisierung mit None in Python

```
my_num = None
```

- Type hinting in Python

```
my_num: int = None
```

Datentypen

```
>>> print(name + " " + birth_year)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

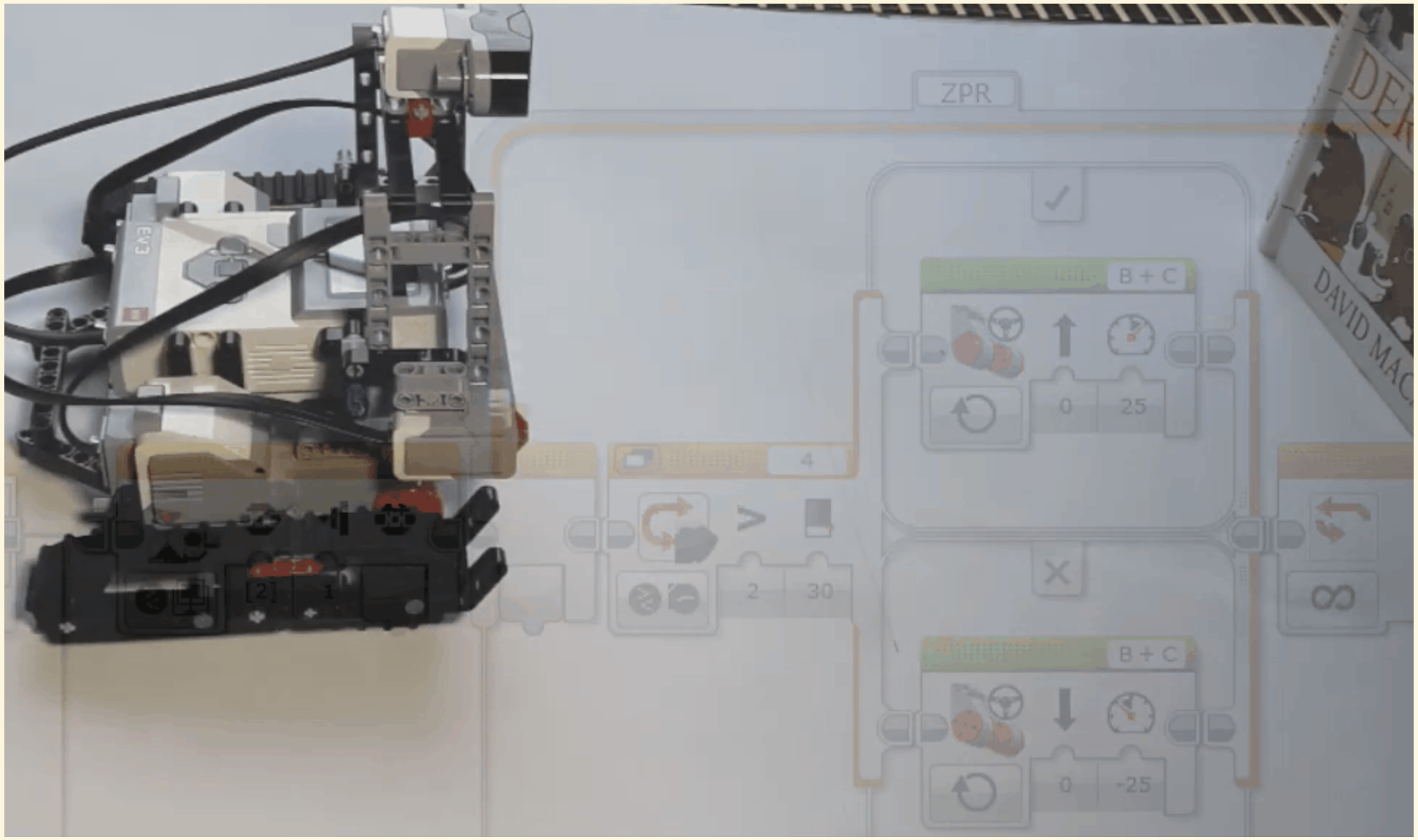
Casting

- Umwandeln des Datentyps einer Variablen
- durch Voransetzen des Typs und Klammern

```
print(name + " " + str(birth_year))
```

Strings Zusammenfügen

- direkt: `first_name + " " + last_name`
- format-statement mit Platzhaltern:
  `"{} {}".format(first_name, last_name)`
- mittels "F-String": `F"{first_name} {last_name} says hello!"`


ZPR
B + C
0
25
4
2
30
[2]
1
B + C
0
-25

✍️ Aufgabe

- Schreiben Sie ein kurzes Programm in der Datei `robot_run.py`
- Dieses speichert `robot_name`, `robot_type`, `distance_to_wall_cm` und `speed_cm_s` in Variablen
- Errechnet `time_to_hit_the_wall_s`
- Gibt Roboter-Name, Typ, sowie Zeit bis Einschlag mittels `print()` aus

Musterlösung - `robot_run.py`

- Erste `Section` in der Datei `robot_run.py`

Datentypen

Boolesche Ausdrücke

- Variablen mit Inhalt `True` oder `False`
- Zuweisung: `game_active = True`
- Vergleich:
  - Gleichheit: `==`
  - Ungleichheit `!=`
  - Größer, kleiner etc.: `>`, `>=`,`<`, `<=`

Boolesche Operationen

- `and`: beide Bedingungen müssen wahr sein
- `or`: mindestens eine Bedingung muss wahr sein
- `not`: Negation

✍ Aufgabe

Passen Sie das Programm so an, dass anzeigt wird, ob sich der Roboter näher als 5 cm zur Wand befindet

Musterlösung - `robot_run.py`

- Zweite `Section` in der Datei `robot_run.py`

Gültigkeitsbereiche

- Anstelle von Klammern werden in Python Einrückungen (whitespace) genutzt, um den Gültigkeitsbereich zu definieren
- Standard für die Einrückung ist vier Leerzeichen bzw. ein Tabstopp
- Scope kann nicht beliebig geöffnet werden

`if`-Bedingungen

- `<conditional_test>` muss in einer booleschen Variable resultieren

```
if <conditional_test>:
  do the first thing
  do the second thing
else:
  do something different
do after the conditional
```

✍️ Aufgabe

- Passen Sie das Programm so an, dass der Roboter durch die Variable `emergency_stop = True` gestoppt wird, wenn er näher als $5\,\mathrm{cm}$ zur Wand ist und dies auch geprinted wird.
- Ist er näher als $10\,\mathrm{cm}$ wird nur eine Warnung geprintet. In allen anderen Fällen wird nur der Abstand ausgegeben.

Musterlösung - `robot_run.py`

- Dritte `Section` in der Datei `robot_run.py`

Datentypen - Listen

- Enthalten eine Folge beliebiger Daten → ähnlich zu Arrays in C
- gekennzeichnet durch eckige Klammern

```
a_list = [1, "zwei" , 3.6]
```

- beliebige Datentypen (`str`, `float`, `int`, etc.) in Liste kombinierbar
- Listen sind mit einem Index durch-nummeriert, dieser startet bei `0`
- genaueres zur Funktionsweise von Listen später

Datentypen - Listen

```
temperature_data = [25.4, 26.1, 24.8, 27.3, 25.9]
```

Manipulation von Listen

- Vollständig in den Python Docs
- Anlegen leerer Listen: `temperature_data = []`
- Aufruf eines bestimmten Elements: `temperature_data[2]`
- Mehrere Elemente als Subliste: `temperature_data[0:2]`
- Hinzufügen: `temperature_data.append(26.7)`
- Entfernen: `del temperature_data[1]`

Datentypen - Listen

- Python versucht möglichst effizient mit Speicherplatz umzugehen
  → Listen werden nur kopiert, wenn dies explizit angegeben wird
- Neuer Name zeigt auf den Speicherort der alten Liste:

```
old_list = [1, 2, 3]
new_list = old_list
print(hex(id(new_list))) #> 0x14189ec2000
print(hex(id(old_list))) #> 0x14189ec2000
```

- Gilt analog auch für verwandte Datentypen (Dictionaries, pandas DataFrames etc.)

Kopieren von Listen

- Explizit mittels `copy()` → `new_list = old_list.copy()`
- Slicing (aller) Elemente → `new_list = old_list[:]`
- `list()`-Konstruktor → `new_list = list(old_list)`

Datentypen - Listen

- Lässt sich auch direkt an den Listen zeigen

Zweiter Name für bestehende Liste

```
list_a = [1,2,3]
list_b = list_a
list_b.append(4)

print(list_b) #> [1,2,3,4]
print(list_a) #> [1,2,3,4]
```

Zweiter Name für eine neue Liste

```
list_a = [1,2,3]
list_b = list_a.copy()
list_b.append(4)

print(list_b) #> [1,2,3,4]
print(list_a) #> [1,2,3]
```

Datentypen - Tupel

- Ähnlich wie Listen → enthalten eine Folge beliebiger Daten
- oft gekennzeichnet durch runde Klammern, oder nur durch Kommas

```
a_tuple = (1, "zwei" , 3.6)
another_tuple = 4, "sieben", 19.3
print(a_tuple[1]) #> zwei
print(another_tuple[2]) #> 19.3
```

- Daten können nicht verändert werden → immutable

```
a_tuple[1] = 10
#> TypeError: 'tuple' object does not support item assignment
```

- Werden häufig für Funktionen verwendet, die mehrere Werte zurückgeben sollen → gibt ein `tuple` zurück, welches die Werte enthält

Datentypen - Dictionaries

- Enthalten Schlüssel-Wert-Paare → Key-Value-Pairs
- Definiert Zuordnung eines Wertes zu einem Schlüssel → Schlüssel muss einzigartig sein
- Praktisch alle Datentypen können als key oder value verwendet werden
- gekennzeichnet durch geschweifte Klammern

```
a_dict = {"key1": "value1", "key2": 35}
print(a_dict["key1"]) #> value1
print(a_dict["key2"]) #> 35
print(a_dict["not_exist"]) #> KeyError: 'not_exist'
```

- genaueres zur Funktionsweise von Dictionaries später

Hilfreiche Funktionen

- Vollständig in den Python Docs
- `keys()`: gibt alle Schlüssel zurück
- `values()`: gibt alle Werte zurück
- `items()`: gibt alle Key-Value-Paare zurück

Kontrollstrukturen

- Beispiel: Iterieren über einen Zahlenbereich

`for`-Schleifen in C

```c
for(int i = 1; i < 11; ++i){
  printf("%d ", i);
}
```

`for`-Schleifen in Python

- die Zählvariable wird dabei automatisch gesetzt

```python
for i in range(0, 11, 1): # eine range ist ein Datentyp für Folgen in Integers
  print(i)
```

- Häufiger wird über iterierbare Datentypen (z.B. Listen, Strings, Dictionaries, etc.) iteriert → Analog zu "for each" in C++

```python
for number in [1,2,3,4,5,6,7,8,9,10]:
  print(number)
```

- `enumerate()` gibt zusätzlich den Index zurück

```python
for index, letter in enumerate(["a","b","c"]):
  print(f"Element {index} is {letter}")
```

Kontrollstrukturen - `for`-Schleifen

- Manchmal wird für einen iterierbaren Typ nur ein Index benötigt → Wert bei `enumerate()` kann ignoriert werden

```
for index, _ in enumerate(["a","b","c"]):
  print(f"Element {index}")
```

- `zip()` erlaubt das Iterieren über mehrere Listen gleichzeitig

```
for a, b in zip([10, 21, 32], ["a", "b", "c"]):
  print(f"{a} is {b}")
#> 10 is a
#> 21 is b
#> 32 is c
```

- `for`-Schleifen erlauben auch die Nutzung des `else`-Statements → wird ausgeführt, wenn die Schleife normal beendet wird

```
for i in range(0, 11, 1):
  print(i)
else:
  print("End of loop")
```

Kontrollstrukturen

- Beispiel: Ausgabe aller ganzen Zahlen von 0-1000, die durch 13 gerade teilbar sind

while-Schleifen in C

```c
int n = 0;
while(n < 1000){
  if(n % 13 == 0){
    printf("%d ", n);
  }
  n++;
}
```

while-Schleifen in Python

```python
n = 0
while(n <= 1000):
  if (n % 13 == 0):
    print(n)
  n = n + 1
```

Kontrollstrukturen - Iterations-/Schleifenabbruch

- In Python analog wie in C/C++ mit `break` und `continue`

`break`

- terminiert Schleife sofort

```
for i in range(5):
  if i == 3:
    break
  print(i)
#> 0 1 2
```

`continue`

- terminiert Iteration & springt direkt in die nächste

```
for i in range(5):
  if i == 3:
    continue
  print(i)
#> 0 1 2 4
```

Stil

- Wie in allen Programmiersprachen gibt es auch in Python Konventionen für den Code-Stil
- Diese sind in Python Enhancement Proposal (PEP) 8 festgehalten
- Linter wie `pylint` oder `flake8` können helfen, den Code-Stil zu überprüfen & einzuhalten

🏆 Hausübung

- Es wurden mit einem triaxialen Beschleunigungssensor Messdaten aufgenommen
- Die Daten sind in einer Liste gespeichert
- Im bereitgestellten Jupyter Notebook sollen Sie die Daten auswerten

1. Umwandlung des Datentyps
2. Bestimmung des Betrags der Beschleunigung
3. Bestimmung der Mittelwerte
4. Glättung mittels gleitendem Mittelwert
5. Bestimmung des Mean Squared Error (MSE) zwischen Original- und geglätteten Daten